NRL Report 7847

On the Need for Fewer Restrictions in Changing Compile-Time Environments

DAVID L. PARNAS

Information Systems Staff
Communication Sciences Division

and

Technische Hochscule Darmstadt, Darmstadt, West Germany

JOHN E. SHORE

Information Systems Staff
Communication Sciences Division

W. DAVID ELLIOTT

Information Processing Systems Branch
Communication Sciences Division

January 10, 1975

NAVAL RESEARCH LABORATORY
Washington, D.C.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NRL Report 7847	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: ON THE NEED FOR FEWER RESTRICTIONS IN CHANGING COMPILE-TIME ENVIRONMENTS	5. TYPE OF REPORT & PERIOD COVERED: An interim report on a continuing NRL problem
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): David L. Parnas, John E. Shore, W. David Elliott	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Research Laboratory, Washington, D.C. 20375	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: B02-18-(XF21-241-021-K211); B02-15-(RF-21-222-401-4356)
11. CONTROLLING OFFICE NAME AND ADDRESS: Department of the Navy, Office of Naval Research, Arlington, Va. 22217	12. REPORT DATE: January 10, 1975
	13. NUMBER OF PAGES: 10
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Compilers
Compiler environments
Macros
Programming Languages

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

During the compilation of a statement, compilers for current programming languages enforce rigid restrictions on changes in the set of associations between names and declarations, that is, on changes in the compile-time environment of the statement. These restrictions inhibit the writing of well-structured, modular programs, because such programs tend to require frequent switching between the environments associated with different modules. With current compilers, arbitrary switching of environments must be handled by means of macros at compile time or deferred until run time.

Continued

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Conventional macros are difficult to use in situations where it is desirable that the macro writer and the macro user be ignorant about each other's programs. When deferred until run time, frequent switching of environments becomes costly owing to the overhead associated with subroutine calls. In order to encourage the compartmentalization of program information and at the same time to allow efficient code generation, we propose the relaxation of present restrictions on changes in the compile-time environment. Specifically, we propose that a group of declarations can be defined as an environment and that arbitrary pieces of program text can state within which environment they are to be compiled. With such a mechanism, program text can be intermixed by means of macros without having to make the relevant declarations global or having to repeat the declarations in places other than the environment's definition.

CONTENTS

ON THE NEED FOR FEWER RESTRICTIONS IN CHANGING COMPILE-TIME ENVIRONMENTS

1. INTRODUCTION AND MOTIVATION

To encourage the compartmentalization of information and at the same time to allow efficient code generation, we propose the relaxation of present restrictions on changes in the compile-time environment. The proposal is an attempt to make the implementation of the program organization and specification techniques described in Refs. 1,2, and 3 practical. In contrast to other work on languages for structured programming, we do not address mechanisms for using this feature to implement any particular type of abstraction (e.g. abstract data types). Our concern is only with returning to the programmer a capability he needs to write well-structured programs. Other mechanisms are left for higher levels.

1.1 Ignorant Programming

The ease of integrating and changing the program described in Ref. 1 was achieved primarily by dividing the task into work assignments (modules) using the "information hiding principle" (Ref. 2). The programmers assigned to any given work assignment must be able to write, test, and improve their programs without any knowledge of the other programs in the system beyond that given by the specifications of those programs (Ref. 3).

1.2 Impracticality of Subroutines

The implementation techniques used for the experiment described in Ref. 1 are not generally suitable for the production of efficient programs. Each module in that experiment was implemented as a collection of Fortran subprograms and all transfers of information and control between modules made use of the subroutine call mechanism. When a program has been decomposed into modules in accordance with strict criteria for the compartmentalization of information, control at run time remains within a module for relatively brief periods of time. Under these conditions, the time spent in executing call and return sequences may be much higher than the remaining processing time. Furthermore, opportunities for optimization may be lost. Because information kept separate at write time is needed together at code generation time and at run time, compartmentalization offers advantages at write time, but potential disadvantages later.

1.3 Macros

When the execution time of the subroutine body is shorter than that of the subroutine call, a standard technique used is to implement the procedure as a macro that inserts

Note: Manuscript submitted November 14, 1974.

the text of the body where the call would have appeared. After invoking a macro, the compiler resumes its processing at the beginning of the inserted text just as if a macro had never been involved. Owing to possible conflicting assumptions about the environment, conventional macros are difficult to use in situations where it is desirable that the macro writer and the macro user be ignorant about each other's programs.

2. COMPILE-TIME ENVIRONMENT

To compile a given statement, compilers make use of associations between names and declarations. The names refer to variables and operators, whereas the declarations, which may be built in, refer to attributes and define data structures and procedures. We refer to these associations as the compile-time environment of the statement.

The restrictions on the way that environments may change as one progresses through program text are quite strong in all compiler languages known to us. In Fortran, the environment is fixed for a compilation unit (program or subprogram). In Algol-like languages, one may augment an environment by adding new associations, which may suppress old ones, and later remove them by terminating the scope of their declaration, which may uncover old ones. However, since the environment changes as the program text is processed sequentially, one may not reenter an environment that has been previously exited. Variables declared in a block are not accessible in any statement that appears after the end of that block in the text.

3. RUN-TIME ENVIRONMENT

The run-time environment of a program is defined by the set of associations between address parts of instructions and physical memory locations. The environment is defined by the address computation algorithm, the contents of any addressing registers, and the contents of those memory locations that may be used in the address computation process.

When we examine the way that the run-time environment of a compiled program may change during execution, we find none of the restrictions mentioned in the previous section. Changes in environment are effected by changing the contents of the registers and tables; there are almost no restrictions on such changes.

4. ENVIRONMENT CHANGES AND MODULAR PROGRAMMING

When modules attempt to hide design decisions from each other, the compile-time environments of separate modules are disjoint. Programs from one module do not include references to data structures or operators that are defined inside another. A transfer of control from one module to another involves a change of environment.

When the transfer between environments is made by a procedure or subroutine call, the environment change is made at run time, when there are no major restrictions on the

environment changes that may occur. Standard procedure call mechanisms make time- and space-consuming environment changes by saving and later restoring the contents of registers and memory locations used in address calculations and certain general registers as well. A compiler that could generate code while making only the necessary changes would require detailed knowledge of both programs simultaneously and would be quite difficult to construct. Even with such a compiler, one would forego opportunities for optimization.

When macros are used as the transfer mechanisms between modules, the compiler can know the characteristics of both the calling and the called code, using only local knowledge about the section of program being compiled. The use of macros can result in relatively efficient transfers between modules because the macro writer can take advantage of his knowledge of precisely what environment changing is necessary.

When design decisions are hidden within modules, the pieces of text inserted by macros from a given source module often communicate by means of variables that the macros have in common but that are not accessible in the surrounding and intervening text where they are inserted by macro expansion. In other words, the macro text must be compiled in an environment that is entirely disjoint from that of the surrounding code but identical to that used in other (possibly quite remote) parts of the program. As discussed in the foregoing, current programming languages do not permit this.

5. AN ENVIRONMENT CHANGING FACILITY

To provide a facility with which pieces of code from completely separate write-time environments are executed without complete changes of run-time environments, the compiler must be capable of transferring between environments at compile time while producing code that appears to have been compiled from statements in a single larger environment. Run-time environment changes can be limited to those necessitated by the availability of addressing registers.

We propose a relatively direct approach. An environment may be defined by surrounding a group of declarations with a "startenvi" statement and an "endenvi" statement, where a "startenvi" or "endenvi" statement consists of the keyword followed by the name and a semicolon. Variables declared in an environment are local to that environment unless declared "accessible". Environments may be declared within environments, and environment names are local to the environment in which they are declared. The outermost set of environment names is known everywhere unless suppressed by reuse of the name. The order in which environment names appear within the description of an environment is irrelevant. The keywords "enter " and "leave " followed by an environment name cause changes of environment.

The "enter " statement invokes the declarations defined within the named environment. The "leave " statement cancels the last "enter ", restoring the previous environment. Each "enter " statement must have a matching "leave " statement. It is important to note that environment changes may be found anywhere: in the middle of statements, in the middle of arithmetic expressions, in the middle of declarations, etc.

Because we have no experience in the use of this simple syntax proposal, we include it not as a specific language design, but as an illustration. When the facility is incorporated in a full language, an appropriate notation can be found that is consistent with the rest of the language.

6. ON RESTRICTING ACCESS

We are not proposing that higher level languages or system implementation languages provide the unrestricted access rights available in assembler languages. On the contrary, we feel that the restriction of access is essential to controlling the structure of programs. We believe, however, that the restrictions in current languages are the wrong restrictions. They are so strong that they force the use of global variables with the result that access-right restrictions are too weak. The proposed environment-changing facility will reduce the need for global variables (Ref. 4).

7. RELATION TO THE PROBLEM OF MACROS

The proposed environment-changing facility can be used to solve the problems of macros previously discussed. In simplistic terms, we hope to gain efficiency by replacing procedure calls with macro calls, and to provide the capability of compiling the names of variables appearing in text produced by a macro in the environment containing the macro definition and not in the environment of the macro call. This is not supported in languages in which, once a macro expansion is produced, the compiler forgets that it was not part of the original text. Using the concept of explicit environment transfer we can surround any macro-produced text with an "enter " and corresponding "leave " so that the compiler will automatically change environments. Further, parameters passed to a macro can be returned and subsequently interpreted in the environment of the caller who passed them. This will allow variable names in the macro environment and actual parameters to conflict without any problems. Conflicting names no longer conflict!

8. AN ILLUSTRATIVE EXAMPLE

The frequent module changes that result from information-hiding techniques can lead to frustrating inefficiencies if an initial call leads finally to something relatively simple like an array reference. In such situations, the environment-switching facility that we have proposed would be particularly useful.

Consider a message-processing system in which, as part of the processing involved in one work assignment, it is necessary to place on a stack message identifiers called "date_time_groups". This function is performed by the following code that appears within a work assignment called module A:

```
integer k;
.
.
.
push(date_time_group(k));
.
.
.
```

The form in which a "date_time_group" is stored or computed is hidden from module A in module B. The writer of module B decides to place messages into a large array called "messages" and to service requests for parts of the message by using macros that return appropriate array references. The following code appears in module B:

```
startenvi Y; integer array messages[1000,1000];
        accessible macro date_time_group(m);
            string m;
            return with 'enter Y messages [300,20+ leave Y' m 'enter Y] leave Y';
endenvi Y;
```

The stack is maintained by yet a third module. The following code appears in module C:

```
startenvi X;
    integer array stack [1:1000];
    integer i;
    accessible macro reset; return with 'enter X i:=0 leave X';
    accessible macro depth; return with 'enter X i leave X';
    accessible macro val; return with 'enter X stack [i] leave X';
    accessible macro pop; return with 'enter X i:=i-1 leave X';
    accessible macro push (it); string it;
        return with 'enter X i:=i+1; stack [i]:=leave X' it;
endenvi X;
```

By invocation of the macros "push" and "date_time_group" the "push" statement becomes

```
enter X i:=i+1; stack [i] := leave X enter Y messages [300,20 +
                             leave Y k enter Y] leave Y;
```

which can then be compiled into efficient code that refers to variables defined in all three modules.

9. AN IMPORTANT UNSOLVED PROBLEM

Use of the proposed environment-changing facilities will produce from a structured representation a program representation with no apparent structure. We consider this completely acceptable only so long as human beings need never study the resulting unstructured program. Whatever structured programming is, its value comes when humans process programs, not when the machines process them. However, run-time errors will force the programmer to study the unstructured program representation resulting from the macro expansions. To solve this problem one must (a) require that every program follow a relatively rigid procedure for handling errors and (b) keep records about the expansion so that when an error is found, it can be associated with the proper higher level constructs. Some remarks about the first problem have appeared in Ref. 5, but both problems require considerable further study. The authors ask communication from those who have suggestions or experience with either problem.

10. ACKNOWLEDGMENTS

We are happy to acknowledge the many useful comments of Dr. James Miller of Intermetrics, Inc. during discussions of the applicability of the concepts discussed in this paper to the CS-4 programming language (Ref. 6). We are also grateful to Professor Dr. H. J. Hoffamann of the Technische Hochscule Darmstadt for insightful comments on an early version of this document.

REFERENCES

1. D. L. Parnas, "Some Conclusions from an Experiment in Software Engineering," Proc. 1972 Fall Joint Comput. Conf. AFIPS vol. 41, p. 325 Nov. 1972.

2. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Commun. ACM vol. 15, No. 12, 1053-1058 Dec. 1972.

3. D. L. Parnas, "A Technique for Software Module Specification With Examples," Commun. ACM vol. 15, No. 5, 330-336 May 1972

4. W. A. Wulf, and M. Shaw, "Global Variable Considered Harmful," SIGPLAN Notices vol. 8, No. 2 (Feb. 1973).

5. D. L. Parnas, "Response to Detected Errors in Well-Structured Programs," Carnegie-Mellon University Technical Report, July 1972.

6. J. S. Miller, C. M. Mikkelsen, J. R. Nestor, B. M. Brosgol, J. T. Pepe, and R. Fourer, CS-4 Language Reference Manual and Operating System Interface, Contract N00123-72-C-1177, NELC, San Diego, Dec. 1973.